Suorituspalveluarkkitehtuurin ja perinteisen sovellusarkkitehtuurin väliset erot

Joni Laurila

Tampereen yliopisto
Luonnontieteiden tiedekunta
Pro gradu -tutkielma
Ohjaaja: Jyrki Nummenmaa
6.3.2018

Tampereen yliopisto
Luonnontieteiden tiedekunta
Joni Laurila: Suorituspalveluarkkitehtuurin ja perinteisen sovellusarkkitehtuurin väliset erot
Pro gradu -tutkielma, 38 sivua
Maaliskuu 2018

Tiivistelmä

Tutkielmassa verrataan suorituspalveluarkkitehtuurin ja perinteisen sovellusarkkitehtuurin eroja API-rajapinnan toteutusta vertailemalla. Vertailua varten on toteutettu kaksi ratkaisua hyödyntäen Amazon Web Service -työkaluja ja Serverless Frameworkiä. Tarkastelu keskittyy pitkälti kehitystyön eroihin, ja tuloksissa pohditaan taloudellisia eroja ja suorityskyjy eroja.

Sisällys

1 Johdanto

Tutkielmassa tarkastellaan suorituspohjaisen arkkitehtuurin (Function-based architecture) ja perinteisen sovellusarkkitehtuurin eroja. Tarkastelu tapahtuu hyödyntäen yksinkertaista jäsenrekisterisovellusta, joka on toteutettu molemmilla teknologioilla. Sovellusta tarkastellaa enemmän luvussa 2, jossa kuvataan yhteisiä teknisiä ratkaisuja, jotka ovat valittu selvittämään vertailua. Sovellus on yksinkertainen, eikä kumpikaan tekniikka pääse täysiin vahvuuksinsa tämän takia. Kuitenkin suurimmat erot pystytään havaitsemaan. Suorituskyvyn vertaaminen tällä sovelluksella olisi hyödytöntä. Suorityskykyä tarkastellaan lähinnä lähteiden pohjalta.

Tutkielmassa käytetään Amazon Web Servicen [Amazon, 2017a] tarjoamia palveluita. Amazon Web Service tarjoaa monia erilaisia pilvipalveluita kansainvälisesti ja on yksi suurimpia toimijoita alalla. Monet muut pilvipalveluiden tarjoajat mahdollistavat hyvin samanlaisen toteutuksen ja myös muiden palveluita olisi voinut käyttää tutkielmassa yhtä hyvin. Näistä esimerkkeinä ovat IBM:n BlueMix [IBM, 2017], Google Cloud Platform [Google, 2017] ja Microsoft Azure Cloud [Microsoft, 2017]. Nämä kaikki palvelut mahdollistavat hyvin samantyylisten arkkitehtuurien käytön ja suorituspohjaiset tilattomat funktiot ovat viimeisimpiä pilvipalveluiden ratkaisuja. Amazon Web Servicen palveluista, jotka ovat käytössä tutkielmassa ovat: AWS EC2, AWS DynamoDB, AWS Lambda, AWS ApiGateway. Näiden lisäksi hyödynnetään Serverless Framework -nimistä työkalua [serverless.com, 2017], joka mahdollistaa AWS Lambdan, AWS ApiGatewayn, AWS DynamoDB:n esittämisen ja julkaisun huomattavasti yksinkertaisemmalla rakenteella kuin Amazon Web Servicen oma CloudFormation. Tämän lisäksi se mahdollistaa tilattomien funktioiden tallentamisen versionhallintaan.

Yksi merkittävämpiä eroja näiden kahden eri sovellusarkkitehtuurin välillä on erityisesti sovelluksen ylläpidossa. Perinteisessä sovellusarkkitehtuurissa usein hyödynnetään joko virtuaalipalvelimia pilvipalvelutarjoajalta tai perinteiseltä käyttöpalvelun tarjoajalta. Usein näiden virtuaalisten ja fyysistenpalvelimien välillä ei näy merkittävää eroa sovelluksen ylläpitäjälle, paitsi toimitusajoissa. Suorituspohjaisessa arkkitehtuurissa ylläpito tapahtuu lähes kokonaan sovelluksen lähdekoodia ylläpitämällä. Palvelimien asetusten ylläpito ja valvonta ulkoistuu sekä piiloutuu sovelluksen ylläpitäjältä palveluntarjoajalle, esimerkkitapauksessa siis Amazon Web Servicen taakse.

Suorituspohjaisissa palveluissa palveluntarjoaja tarjoaa palvelua, jossa sovelluksen toiminta on hajautettu tilattomiin funktioihin. Tämä mahdollistaa myös teo-

riassa rajattoman skaalautumisen, koska aina toimintoja käytettäessä kutsutaan uutta tilatonta funktiota. Tämä käynnistää uuden funktion ja näitä funktioita voi olla rinnakkain useita samaan aikaan käynnissä. Rinnakkaisuuden avulla päästään eroon perinteisen mallin palvelimella ajettavan sovelluksen skaalautumis ongelmista, sillä korkean kuorman alla resurssit palvelimella loppuvat kesken. Vastaavasti kun funktiota ei tarvita, sitä ei kutsuta eikä se vie mitään resursseja. Tämä mahdollistaa resurssien säästämisen hiljaisempina hetkinä. Resurssien säästäminen näkyy usein suoraan ylläpitokulujen vähentymisenä.

2 Jäsenrekisterijärjestelmän kuvaus

Järjestelmä mallintaa yksinkertaista jäsenrekisterin REST (Representational State Transfer) -arkkitehtuuria hyödyntäen UTA Sportin kaltaiselle järjestölle, jolla on monia alajärjestöjä. UTA Sport seuraa ja ylläpitää kaikkien alajärjestöjen jäsenyyksiä. Järjestelmään on pääsy vain muutamalla UTA Sportin ja sen alajärjestöjen hallitusten edustajilla. Tutkielmassa käytetty versio on hyvin pelkistetty toteutus mahdollisesta ratkaisusta, jota voitaisiin käyttää tähän tarkoitukseen. Tässä kyseisessä versiossa ei ole huomioitu käyttäjienhallintaa ja tietoturva on jäänyt vähäiselle huomiolle. Esimerkiksi taustapalvelulle tunnistautumista ei ole toteutettu ollenkaan.

2.1 Rajapinnan yleinenkuvaus

Rajapintaan ei kuulu käyttöliittymää, vaan pelkästään jäsenrekisterin ja sovelluslogiikan sisältävä sovellusrajapinta. Rajapintaan ei ole toteutettu kirjautumista ja pääsynhallintaa, koska kyseinen versio ei ole tarkoitettu tuotantokäyttöön. Samasta syystä rajapinnalle ei ole myöskään huomioitu mitään valvontakeinoja sen toimivuuden valvomiseksi.
Sovelluksia pystyy hallitsemaan API-rajapinnan kautta. Rajapinnan kautta voi lisätä, poistaa ja muokata jäseniä. Jäsenrekisteriin tallennetaan kaikkien jäsenten yhteystiedot ja tiedot seuroista, joihin jäsenellä on jäsenyys. Jäsenrekisteri on toteutettu yksinkertaistamisen takia dokumenttikantana ja ilman relaatioita. Rajapinta kuvataan tarkemmin kohdassa 2.2.
Tutkielman aiheena oleva sovellusrajapinta on toteutettu kahdella eri tekniikalla. Ensimmäinen versio on toteutettu perinteisempää sovellusarkkitehtuuria hyödyntäen, jossa taustapalvelut ja sovelluslogiikka ovat yksi palvelu, joka pyörii virtuaalipalvelimella. Toinen on toteuttu hyödyntäen tilattomia funktioita, jotka toteuttavat sovelluslogiikan ja pyörivät hyödyntäen Amazon Web Service Lambda palvelua.
Testauksesta on toteutettu vain yksikkötestit apufunktioille, mutta tietokantatestejä ei ole toteutettu. Tarkoituksena on tunnistaa ratkaisujen väliset erot ohjelmistokehitystä ja ylläpitoa helpottavien yksikkötestien muodossa.

2.2 API-rajapinnan kuvaus

Rajapinnan kautta hallitaan jäsenrekisteriin tallennettuja jäseniä. Rajapinnalle ei ole toteutettu kokonaisvaltaista testausta (end-to-end) johtuen järjestelmän

yksinkertaisuudesta.

Rajapinnan kautta pystyy:

- Lisäämään jäsenen
- Poistamaan jäsenen
- Muokkaamaan jäsentietoja
- Lisäämään jäsenelle jäsenyyden alaseuraan
- Poistamaan jäseneltä jäsenyyden alaseurasta
- Listaamaan koko jäsenrekisterin
- Listaamaan tietyn alaseuran jäsenrekisterin.

Käyttöliittymää palvelulle ei toteuteta, koska sitä ei tarvita testausta varten. Kaikki tarvittavat kutsut pystytään testaamaan käyttämällä Postman-sovellusta [Technologies, 2017]. Postman on tarkoitettu API-rajapintojen kehittämistä varten. Molemmat palvelut toteuttavat rajapinnan samoilla säännöillä vain web-osoite on eri.

2.3 Tietorakenteen kuvaus

Tietorakenteessa hyödynnetään dokumenttikannan rakennetta, jossa jäsen on yksi dokumentti. Dokumentissa on seurajäsenyyskenttä, johon tallennetaan jokaisen seuran jäsenyys ja sen aktiivisuus. Listauksessa 1 esitetään esimerkki jäsendokumentista. Tietokannassa pääavaimena toimii sähköpostiosoite, koska se on uniikki jokaisella yksittäisellä käyttäjällä. Tuotantototeuksessa olisi hyvä käyttää uniikkia id:tä käyttäjien pääavaimena, koska tällöin voidaan varmistaa uniikit pääavaimet.

```
{
  "email": "example@email.com",
  "name": "John Doe",
  "phone": "+358123456",
  "address": "Example street 2, 33100, TAMPERE",
  "memberships": [
    { "team": "utasport", "expires": "always" },
    { "team": "apa", "expires": "30.03.2018"},
    { "team": "random", "expires": "01.01.2017"}
  ]
}
```

Listaus 1: Esimerkki jäsendokumentista

Molemmat toteutukset hyödyntävät samaa versiota tietokannasta ja samaa tietokantaa. Tietokannan kanssa on hyödynnetty AWS DynamoDBD -toteutusta, joka on dokumenttikanta. Kyseinen dokumenttikanta poistaa mahdollisuuden relaatioiden käyttöön, mutta jäsenrekisterin yksinkertaisessa ratkaisussa tämä ei haittaa. Lisäksi kanta ei ole tutkielmassa tarkastelun kohteena. Ongelmana isoissa tietokannoissa AWS DynamoDB:n käytön kohdalla on, että tietokantaan tehdään aina kaiken aineiston haku, kun haetaan jotain tiettyä riviä. Tämä on isossa tietokannassa tehoton ratkaisu. Tuotantokäytössä toteutettaisiin varmasti tietokannan relaatiokantana, eikä dokumenttikantana. [Amazon, 2017d]

2.4 Taustapalvelun yleinen kuvaus

Tutkielman aiheena olevien taustapalveluiden vertaaminen tapahtuu edellä esitettyjen tietokannan ja rajapinnan avulla. Taustapalvelu toteuttaa kaikki ominaisudet, joita rajapinta tarvitsee. Taustapalvelu tarjoaa API-palvelurajapinnan, jota käyttöliittymä voi hyödyntää. Taustapalvelu hoitaa tietokannan muokkaamisen API-rajapinnan välityksellä tulevien komentojen perusteella. Taustapalvelu toteutetaan toisessa esimerkissä suorituspohjaisella ratkaisulla hyödyntäen AWS Lambda -funktioita ja AWS API Gateway -palveluita. Toisessa esimerkissä se toteutetaan hyödyntäen virtuaalikoneratkaisua.

REST-rajapinta toteuttaa:

- Jäsenten haun
- Jäsenten muokkaamisen
- Jäsenten luomisen
- Jäsenten poistamisen
- Jäsen aktiivisuuden tarkastus (jäsenmaksu).

Testausta varten käytetään Mochajs-nimistä kirjastoa. Mochajs mahdollistaa JavaScriptin yksikkötestaamisen ja helpottaa yksikkötestien toteuttamista merkittävästi. [Mochajs, 2017]

3 Suorituspohjainen arkkitehtuuri

Luvussa tutustutaan käytettyyn suorituspohjaisen arkkitehtuurin ratkaisuun. Aluksi käydään läpi käytetyt teknologiat ja tietokannan luonti. Tietokanta luodaan vain tämän toteutuksen yhteydessä, koska molemmat ratkaisut tutkielmassa hyödyntävät samaa tietokantaa ja suorituspohjaisen ratkaisun kanssa sen luominen oli nopeampaa ja helpompaa. Luvun loppupuolella esitellään sovelluslogiikan toteutus funktiot esittelemällä.

3.1 Käytetyt teknologiat

Suorituspohjaisessa ratkaisussa hyödynnettiin Serverless Framework -työkalua Amazon Web Servicen palveluiden käytön helpottamiseksi. Serverless Framework mahdollistaa kohtuullisen yksinkertaisen yaml-tiedoston tekemisen, joka ohjaa Amazon Web Service -resurssien rakentamista automaattisesti. Serverless Frameworkin ansiosta ei tarvitse käyttää Amazon Web Servicen tarjoamaa web-käyttöliittymää. Konfiguraation ja sen muutokset saadaan helposti versiohallintaan talteen. Listauksessa 2 esitellään, miltä näyttää serverless.yml kokonaisuudessaan. YAML-tiedoston avulla Serverless Framework asettaa kaikki tarvittavat Amazon Web Service -palvelut haluttuun muodoon. Konfiguraatioasetuksia esitellään tarkemmin listauksessa 3 ja resurssien hallintaa listauksessa 4.

Amazon Web Service asetuksen ja palveluiden konfiguroiminen tapahtuu pääasiassa tarjoaja- (provider) ja resurssi-osioissa (resources). Funktio-osiossa (Functions) esitellään funktiot ja niiden rajapinnat. Funktioista enemmän kohdissa 3.6–3.9. Listauksessa 3 esitellään palvelussa käytetty tarjoaja ja palvelu eli palvelun nimi. Tarjoajaosuudessa esitellään tiedostolle yleisiä asetuksia ja voidaan määritellä AWS-ympäristöön käyttöoikeuksia. Siinä muun muassa määritellään, minne Amazon Web Servicen pilviympäristöön palvelu pystytetään (region) profiili, jonka Amazon Web Service tunnuksia käytetään, Lambda-funktioille varattu muistinmäärä (vaikuttaa hintaan) ja minkälaiset oikeudet Lamba-funktioille annetaan ja mihin resurssiin (iamRoleStatements). Listauksessa 3 on myös esitelty mukautettu (custom) asetus, jolla on mahdollista esitellä muuttujia konfiguraatiolle. Muuttujilla voi hallita usein toistuvia nimiä konfiguraatiossa. Amazon Web Service käyttää IAMRole:a käyttöoikeuksien hallintaan, niiden avulla voi rajata eri palveluiden tai käyttäjien oikeuksia ja/tai tehdä eri asioita Amazon Web Service -ympäristöissä [Amazon, 2017e]

```
service: {project name}
provider:
  name: aws
  runtime: nodejs6.10
  region: eu-west-1
...
functions:
  getMembers:
    handler: handler.get
...
resources:
  Resources:
    MembersTable:
...
```

Listaus 2: Kooste serverless.yaml -tiedoston sisällöstä kokonaisuudessa.

```
service: {project name}
provider:
  name: aws
  runtime: nodejs6.10
  region: eu-west-1
  profile: {profile}
  memorySize: 128
  iamRoleStatements:
    - Effect: Allow
      Action:
        - dynamodb:*
      Resource: "arn:aws:dynamodb:eu-west-1:*:table/${self:custom.tablename}"
custom:
  tablename: {dynamodb table name}
```

Listaus 3: Esimerkki Amazon Web Servicen tarvitsemasta konfiguraatiosta.

Listauksessa 4 luodaan DynamoDB-resurssi. Tässä syntaksi on Amazon Web Servicen käyttämää YAML-syntaksia. Käytännössä siis tässä esitellään taulu, joka luodaan, sen nimi ja avain. Lisäksi määritellään, kuinka monta samanaikaista lukijaa ja kirjoittajaa taululle varataan, mikä vaikuttaa hintaan.

```
resources:
  Resources:
    MembersTable:
      Type: AWS::DynamoDB::Table
      Properties:
        TableName: ${self:custom.tablename}
        AttributeDefinitions:
          - AttributeName: email
            AttributeType: S
        KeySchema:
          - AttributeName: email
            KeyType: HASH
        ProvisionedThroughput:
          ReadCapacityUnits: 1
          WriteCapacityUnits: 1
```

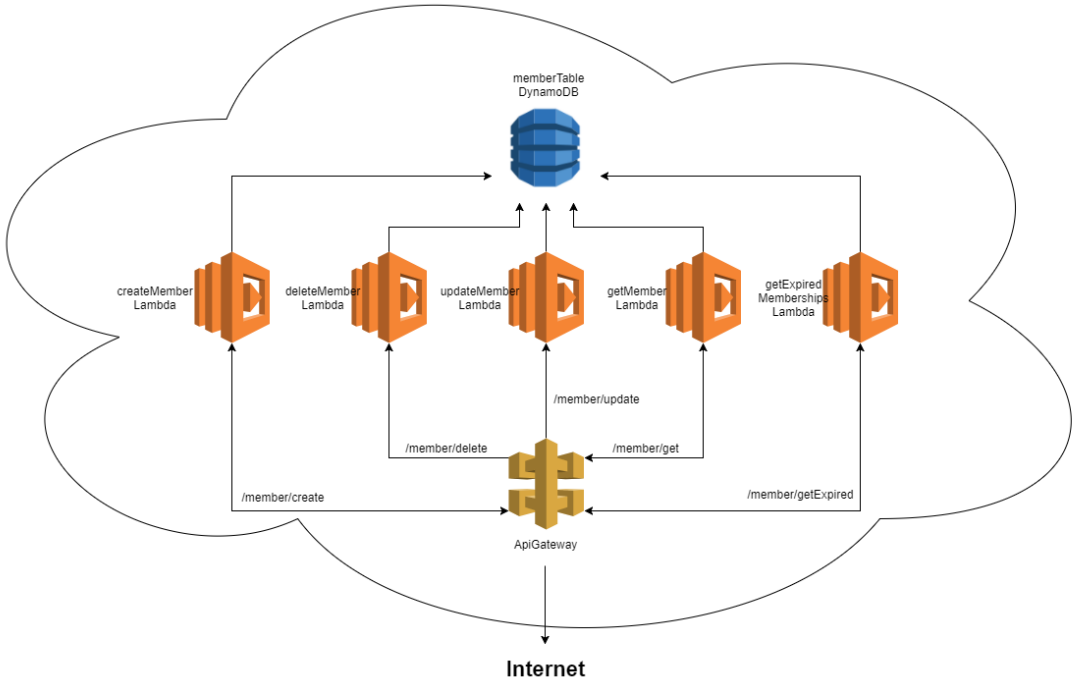
Listaus 4: Esimerkki DynamoDB -resurssin esittämisestä serverless.yaml-tiedostossa.

Muita käytettyjä työkaluja ovat versionhallintajärjestelmä GIT ja yksikkötestien tekemiseen Mocha.js. API-rajapinnan testauksessa hyödynnettiin niin Serverless Frameworkin komentorivityökaluja kuin myös Postman-sovellusta, joka mahdollistaa graafisesta käyttöliittymästä API-kutsujen tekemisen. API-testaus tehtiin käsin. Itse sovelluslogiikka on ohjelmoitu käyttäen JavaScript-kieltä.

3.2 Rakenteen kuvaus

Suorituspohjaisen toteutuksen arkkitehtuurikuva on hieman liioiteltu ja jokainen Lambda-funktio on eriytetty omaksi kuvakseen. Tämä on tehty erojen havaitsemisen helpottamiseksi. Kuvassa 3.1 esitetään kaikki resurssit, jotka projektin serverless.yaml luo ja ylläpitää. AWS ApiGateway tarjoaa jokaiselle Lambda-funktiolle

oman polun ja mahdollistaa näin funktioiden kutsumisen. Lambdafunktiot ovat yhteydessä AWS DynamoDB:hen ja hakevat tai muokkaavat tietoa siellä. Saatuaan vastauksensa funktio palauttaa sen AWS ApiGatewayn kautta kutsujalle.



Kuva 3.1 Suorityspalveluarkkitehtuurikuva

3.3 Julkaisu ja päivittäminen

Serverless Framework tekee Amazon Web Serviceen palvelun julkaisusta todella yksinkertaista. Se on mahdollista hyödyntämällä komentorivityökalua ja yksinkertaista komentoa. Listauksessa 5 esitelty komento paketoi ja julkaisee koko serverless.yaml-tiedoston määrittämän palvelun Amazon Web Servicessä. Tähän kuluva aika riippuu siitä, kuinka isot lähdekoodit sovelluslogiikan osalta siirretään palveluun. Projektin tapauksessa aikaa kului noin 30–60 sekuntia/päivitys.

```
$ serverless deploy
```

Listaus 5: Komentorivikomento, jolla julkaistaan koko serverless.yaml ja lähdekoodit Amazon Web Serviceen.

Listauksessa 6 esitetyllä komennolla pystyy julkaisemaan vain yhden halutun funktion Amazon Web Servicessä, mutta tämä vaatii, että ainakin kerran on tehty listauksen 5 mukainen komento. Kuitenkin tämä on hyödyllinen komento, jos on tehnyt vain tiettyyn funktioon muutoksia. Jos tekee muutoksia serverless.yaml-tiedostoon, muutokset eivät siirry tämän komennon avulla AWS-ympäristöön. Komennon suorittamissa ja funktion päivittämisessä listauksen 6 mukaisella komennolla menee noin 2–5 sekunttia, eli huomattavasti vähemmän kuin listauksen 5 mukaisella komenolla. Tämä mahdollistaa hyvinkin nopean ja kettäreän kehitysprosessin, jossa pienten koodimuutosten tekeminen ja testaaminen tapahtuu nopeasti. Eri versioita voi hallita konfiguroimalla stagen serverless.yaml-tiedoston provider-osaan, jolloin tuotantoversiota ei tarvitse päivittää ennen kuin uuden version toimivuus on varmistettu.

```
$ serverless deploy function --function {function-name}
```

Listaus 6: Komentorivikomento, jolla julkaistaan vain haluttu funktio Amazon Web Serviceen.

Lambda-funktioiden toimivuutta on tärkeää seurata tai asettaa automaattivalvonta seuraamista varten. Seuranta ja valvonta tapahtuvat hyödyntäen AWS CloudWatchia, jonne Lambda-funktiot kirjoittavat tapahtumalogiaan. AWS CloudWatch tarjoaa suoraan mahdollisuuden tehdä hälytyksiä, kun jotain määriteltyä tapahtuu AWS Lambdan lokeissa. Näitä hälytyksiä on mahdollista hallita AWS Web Consolesta. Hälytyksiä ei toteutettu projektissa, mutta niiden toteuttaminen on yksinkertaista käyttöliittymän kautta. [Amazon, 2017c]

3.4 API-rajapinta

API-rajapinta on toteutettu hyödyntämällä Amazon Web Service ApiGateway -palvelua. Palvelun avulla on helppo tarjota esimerkiksi AWS Lambda -funktiot saataville julkisesti ja helposti käytettäviksi. Lambda-funktioita voi kutsua suoraankin, mutta silloin kutsujalla pitää olla tietyt oikeudet, joita voi hallita AWS IAMRole:lla. AWS ApiGatewayn avulla pystyy kutsumaan Lambda-funktioita ilman erillisiä oikeuksia tai voi käyttää monia muita mahdollisia tunnistautumisia, kuten APIKey:tä tai toteuttaa räätälöidyn tunnistautumisen esimerkiksi OAuthia hyödyntäen. AWS ApiGateway tuotetaan serverless.yaml:ssä jokaiselle

funktiolle erikseen. Listauksessa 7 näytetään esimerkki siitä miten helloworld-funktiolle asetetaan AWS ApiGateway:n avulla API-rajapinta. Tämä tapahtuu antamalla funktiolle tapahtumien (events) alle AWS ApiGateway -tunniste http ja asettamalla tälle polku (path) ja metodi (method). [Amazon, 2017b]

```
functions:
  helloworld:
    handler: handler.hello
    events:
      - http:
          path: say/hello
          method: get
```

Listaus 7: Yksinkertainen esimerkki ApiGatewayn toteuttamisesta funktiolle.

Tapahtumat on serverless-frameworkin käyttämä tunnus Lambda-funktion herättävälle toiminnolle se voi olla http-kutsu (AWS ApiGatewayn kautta), AWS S3 -ämpärissä tapahtunut muutos, AWS CloudWatch lokitustapahtuma tai jokin muu Amazon Web Servicen tarjoama palvelu Lambda-funktion herättämiseen.

3.5 handler.js funktioiden hallinta

Serverless-frameworkin käyttämässä serverless.yaml:ssa mainitaan jokaisen funktion kohdalla käsittelijä (handler), joka merkitsee sitä mitä koodia tapahtuman laukaisema lambda-funktio alkaa suorittamaan. Handler-sanan jälkeinen osuus kertoo tiedoston ja funktion kyseisestä tiedostosta, joka on osa koodia mitä lambdan tulee alkaa suorittamaan. Toteutuksenssa käytetty hander.js on kokonaisuudessaan esitetty listauksessa 8. Kyseinen tiedosto on myös samalla funktion ensimmäinen paikka, josta Lambda lähtee ajamaan koodia. Tiedostoa ennen ei siis voi olettaa Lambdan tietävän mitään koodista.

```
'use strict';

const AWS = require('aws-sdk');
const db = new AWS.DynamoDB.DocumentClient({reqion: 'eu-west-1'});
const Member = require('src/member');
const member = new Member(db);

module.exports.get = (event, context, callback) => {
  const t = (event.body === undefined) ? '*' :
   ↪  JSON.parse(event.body).team;
  member.getTeam(t, callback);
};

module.exports.update = (event, context, callback) => {
  const m = (event.body === undefined) ? 'No member information' :
   ↪  JSON.parse(event.body);
  member.update(m, callback);
};

module.exports.create = (event, context, callback) => {
  const m = (event.body === undefined) ? 'No memberInformation' :
   ↪  JSON.parse(event.body);
  member.create(m, callback);
};

module.exports.delete = (event, context, callback) => {
  const r = (event.body === undefined) ? 'No information to
   ↪  remove' : JSON.parse(event.body);
  member.delete(r, callback);
};

module.exports.getExpired = (event, context, callback) => {
  member.getExpired(callback);
};
```

Listaus 8: Handler.js tiedostossa esitellään kaikki funktiot, jotka AWS Lambda -funktiot toteuttavat.

Funktioita varten voi toteuttaa useita handler-tiedostoja, mikä onkin hyvin suotavaa ylläpitotyön helpottamiseksi. On hyvä jakaa API-rajapinta erillisiin loogisiin kokonaisuuksiin, kuten osapolkuihin. Esimerkiksi jos haluttaisiin lisätä erillinen alaseurojen hallinta rajapinnan kautta. Olisi silloin API-polku "/team/*", jolloin kaikki team-polun funktiot on hyvä laittaa omaan tiedostoon.

3.6 Tilattomien funktioiden esittely

Jäsenten listaaminen tapahtuu lähettämällä AWS ApiGatewaylle HTTP Post-kutsu, joka voi sisältää team-rajoittimen. Rajoittimen tehtävä on valikoida joukkueet, joiden jäsenet halutaan listata. Jos rajoitinta ei laiteta, palauttaa funktio kaikki rekisteröidyt jäsenet. Listauksessa 9 esitellään Serverless-frameworkin käyttämä tieto siitä miten rakentaa getMembers funktio Amazon Web Service-ympäristöön.

```
getMembers:
  handler: handler.get
  events:
    - http:
        path: member/get
        method: post
```

Listaus 9: GetMembers-funktion esittely serverless.yaml-tiedostossa.

Jäsenten lisääminen järjestelmään tapahtuu JSON REST -kutsuna, jossa hyötykuormana välitetään JSON-objekti AWS ApiGatewaylle. Listauksessa 10 esitellään esimerkki JSON-objektista, jolla voi luoda uuden käyttäjän.

```
{
  "email":"postman@email.com",
  "name":"Post Man",
  "phone":"+358123456",
  "address":"Postia,33100,TAMPERE",
  "memberships":
  [
    {
      "team":"utasport",
      "expires":"1.1.2018"
    }
  ]
}
```

Listaus 10: Esimerkki jäsenen lisäämisessä käytetystä JSON-objektista.

Listauksia 9 ja 11 verratessa huomataan, että niissä suurin osa on toistoa. Erot tulevat vain REST-rajapinnan polussa tapahtuman polkuosiossa ja lambdan toteuttavan funktion kohdalla eli käsittelijäosiossa.

```
createMember:
  handler: handler.create
  events:
    - http:
        path: member/create
        method: post
```

Listaus 11: CreateMembers-funktion esittely serverless.yaml-tiedostssa.

Ainoa lambda-funkioiden esittelyista merkkittävästi eroava on GetExpiredMemberships-funktion esittely, jossa luodaan AWS ApiGateway polun päähän HTTP GET -metodia hyödyntävä rajapintakutsu. GetExpiredMemberships-funktion luonti on esitetty listauksessa 12.

```
getExpiredMemberships:
  handler: handler.getExpired
  events:
    - http:
        path: member/getExpired
        method: get
```

Listaus 12: GetExpiredMemberships-funktion esittely serverless.yaml-tiedostossa.

Jäsenen poisto toimii antamalla poistettavan jäsenen sähköpostiosoite JSON-objektina rajapinnalle, ja tämän jälkeen herätellään deleteMember-lambda, joka poistaa käyttäjän ja sen tiedot tietovarastosta. Jäsenen muokkaaminen toimii myös hyvin samalla tavalla kuin uuden jäsenen luonti; siinä annetaan membership-taulukon sijaan vain yksittäinen jäsenyys. Jäsenen muokkaaminen on siis käytännössä tarkoitettu henkilön seuratietojen päivittämistä varten. Listauksessa 13 on esitetty esimerkki JSON-objektista, jolla päivititetään jäsentietoja.

```
{
  "email":"example@email.com",
  "name":"John Doe",
  "phone":"+358123456",
  "address":"Example street 2, 33100, TAMPERE",
  "team":"apa","expires":"always"
}
```

Listaus 13: Esimerkkisyöte, jolla päivitetään John Doen jäsenyystiedot joukkueeseen APA.

Suorituspohjaisessa ratkaisussa lähdekoodin suoritus on hyvin yksinkertainen. Listauksessa 14 on esitetty erikseen, miten getMembers-funktion toteuttava AWS Lambda -funktio on esitelty koodissa. Listauksessa 14 esitetyn koodin avulla Lambda-funktio osaa herättää ja välittää tiedot listauksessa 15 esitetylle funktiolle. Funktio on esitelty Member.js-tiedostossa ja se käyttää myös filterTeams-funktiota, joka on esitelty listauksessa 16.

```
8  module.exports.get = (event, context, callback) => {
9    const t = (event.body === undefined) ? '*' :
       ↪  JSON.parse(event.body).team;
10   member.getTeam(t, callback);
11 };
```

Listaus 14: Get-funktion esittely, handler.js tiedostosta.

```
8    /*
9    * Will get all members of asked team, if team is ´*´ will
       ↪  return all members.
10   */
11   getTeam(team, callback) {
12     const params = {
13       "TableName": "jevli-gradu-members"
14     };
15
16     this.db.scan(params, (err, data) => {
17       if (err) {
18         callback(err);
19       } else {
20         callback(null, this.createResponse(200, "Get members",
             ↪  this.filterTeams(data.Items, team)));
21       }
22     });
23   }
```

Listaus 15: Get-funktion toteuttava osuus lähdekoodista.

```
289   /*
290    * Filter memers what are not part of wanted team. If team is
      ↪  ´*´ will return all members.
291    */
292   filterTeams(data, team) {
293     let result = [];
294     data.forEach( (member) => {
295       if (member.memberships.length === 0 && team === '*') {
296         result.push(this.makeMemberRecord(member, {"team": "",
            ↪  "expires": ""}));
297       } else {
298         member.memberships.forEach( (membership) => {
299           if (membership.team === team || team === '*') {
300             result.push(this.makeMemberRecord(member,
                ↪  membership));
301           }
302         });
303       }
304     });
305     return result;
306   }
```

Listaus 16: Get-funktiossa käytettävä filterteams funktion lähdekoodi.

4 Perinteinen sovellusarkkitehtuuri

Tässä luvussa tutustutaan perinteisen sovellusarkkitehtuurin ratkaisuun. Aluksi käydään tarkemmin läpi käytetyt teknologiat ja ratkaisun rakenne. Tietokantaa ja sovelluksen logiikkaa ei käydä yksityiskohtaisesti läpi.

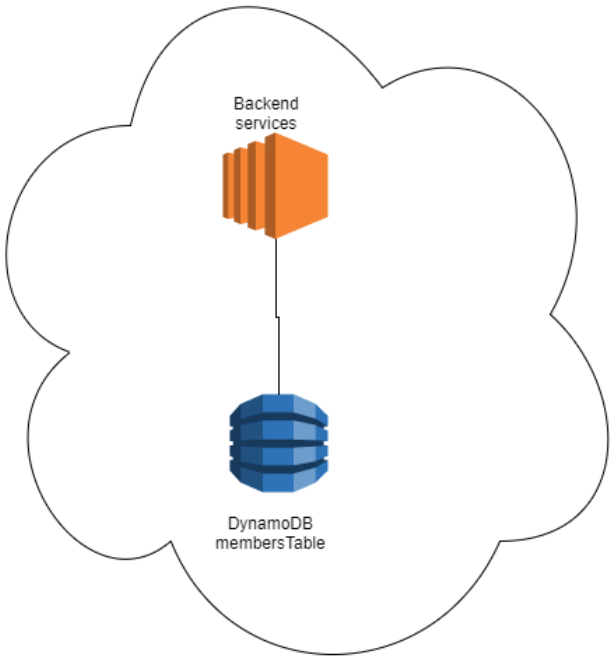
4.1 Käytetyt teknologiat

Perinteistä sovellusarkkitehtuuria varten toteutettiin sovelluslogiikka hyödyntäen Express.js-nimistä sovelluskehystä [StrongLoop/IBM, 2017]. Express mahdollistaa nopeasti tehtävän NodeJS-pohjaisen taustapalvelusovelluksen, eikä ota kantaa juuri mihinkään muuhun. Sovelluksen käyttöönotto ja ajaminen olisi mahdollista lähes kaikilla virtuaalipalvelinvaihtoehdoilla ja sovelluksen voisi paketoida esimerkiksi Docker-imageen, jota voisi ajaa Linux-pohjaisissa ympäristöissä ilman suurempia valmisteluja.
Itse sovellusta ei kuitenkaan laitettu ajoon AWS EC2-ympäristöön alkuperäisen suunnitelman mukaan, koska tarvittavat tulokset pystyttiin tunnistamaan paikallisesti palvelua suorittamalla. Perinteisen sovelluskehityksen etuja onkin, että sovelluksia pystyy ajamaan tuotantoympäristöä vastaavissa ympäristöissä paikallisesti kohtuullisen helposti. Expressjs otettiin käyttöön hyödyntäen NodeJS:n NPM-pakettienhallintasovellusta. Tämän kautta taustapalvelun ainoa ja merkittävä asetustiedosto on package.json.

4.2 Rakenteen kuvaus

Perinteinen sovellusarkkitehtuuri on hyvinkin helppoa kuvata näin yksinkertaisessa palvelussa. Se sisältää tietokantaratkaisun, eli tässä tapauksessa AWS DynamoDB:n, jonne tallennetaan kaikki tieto. Lisäksi se sisältää taustapalvelun eli sovelluspalvelimen. Sovelluspalvelimella pyörii itse sovellus, joka tarjoaa API-rajapinnan, jonka kautta palvelu on tavoitettavissa internetistä. Tämän kaltainen ”monoliitti”arkkitehtuuri on hyvinkin suosittu pienten palveluiden toteutuksessa. Vaihtoehtoinen malli olisi ”hajautettu monoliitti”, eli taustapalvelusta olisi irroitettu pienempiä kokonaisuuksia omiin prosesseihin tai jopa omille palvelimille. Kuitenkin hajautetussa monoliitissä käytetään edelleen yhtä tietokantaa. Kolmas vaihtoehto perinteiselle sovellusarkkitehtuurille olisi mikropalveluarkkitehtuuri, jossa usea palvelu toimii toisistaan erillisinä palveluina, mutta keskustelevat keskenään muodostaen yhden isomman. Mikropalveluarkkitehtuurissa jokaisella mikropalvelulla on oma tietovarastonsa. Mikropalveluarkkitehtuurilla toteutettu-

na tämä palvelu olisi siis todennäköisesti saannut parikin kantaa, jäsenrekisterin ja joukkuerekisterin. Sitten jäseniin ja joukkueisiin jakautunet sovellukset olisivat sitten keskustelleet keskenään muodostaen yhden kokonaisuuden.
Tutkielmassa käytetty vertailutoteutus perinteisestä sovellusarkkitehtuurista noudattaa siis monoliittiratkaisua, jossa koko palvelu on toteutettu yhteen sovellukseen yhdelle sovelluspalvelimelle, kuten kuvassa 4.1 on esitetty.



Kuva 4.1 Perinteinen sovelluspalveluarkkitehtuuri.

4.3 Julkaisu

Johtuen yksinkertaisesta ratkaisusta ja siitä, että toteutuksessa ei käytetty erillisiä julkaisutyökaluja, on sovelluksen julkaisu käsityötä. Tästä johtuen julkaisu tapahtuu seuraavasti:

1. Kirjaudutaan virutaalipalvelimelle ssh-yhteyden avulla

2. Asennetaan tarvittavat työkalut (NodeJS, npm ja git)

3. Kloonataan sovellus repository-palvelimelle

4. Ajetaan npm install -komento sovelluksen juurihakemistossa

5. Konfiguroidaan palvelin käynnistämään sovellus aina mahdollisen kaatumisen jälkeen

6. Käynnistettään palvelu.

Yllämainitut välivaiheet voisi muuttaa yhdeksi ansible scriptiksi, jolla olisi mahdollista julkaista yhdellä kommennolla koko sovellus. Tämä laskisi merkittävästi ylläpidossa ja julkaisussa mahdollisesti tapahtuvien virheiden todennäköisyyttä. Ylläpidon kannalta perinteinen malli on hyvinkin yksinkertainen, kun julkaisutyökaluja on käytössä. Kehitystä voi tehdä täysin paikallisessa ympäristössä ilman virtuaali- tai fyysisiä palvelimia, ja muutoksia voi testata ilman verkkoyhteyttä ulkomaailmaan. Tietenkin paikallisesti testaamisessa nousee helposti esiin ongelmia siitä, että kehitysympäristö ja tuotantoympäristö eivät ole samanlaisia. Tämänlaista tilannetta varten on hyvä harkita virtualisointityökaluja, jotka mahdollistavat yhtenäiset ympäristöt. Esimerkiksi Docker toimii mahdollisena virtuaalisovelluksentyökaluna.

4.4 Sovellusloogiikan esittely

Sovelluksen toteutus on hyvinkin yksinkertainen, minkä mahdollistaa expressjs. Listauksessa 17 on app.js-tiedoston koodi, jolla konfiguroidaan expressjs:n toimintaa. Alussa kerrotaan, mitä moduuleita tarvitaan, ja luodaan sovellus (app).

```
1 const express = require('express')
2 const app = express()
3 const aws_router = require('./routes/aws')
4
5 app.use('/aws', aws_router)
6
7 app.listen(3000, () => { console.log('Express server working')})
```

Listaus 17: App.js-tiedosto, jolla ohjataan sovelluksen toimintaa.

Listauksen 17 rivillä 5 käsketään sovellusta ohjaamaan /aws-polkuun tuleva liikenne aws-reitittimelle (router), joka on määritelty rivillä 3. Rivillä 7 käsketään

sovellusta kuuntelemaan porttia 3000. Listauksessa 18 on esitelty kutsujen reititys /aws-polun takaa, eli API-rajapinnan toteutus. Tämä tiedosto siis käytännössä määrittelee sen, miten /aws-polkuun tuleva liikenne käsitellään. Tiedoston rivimäärän takia siitä on poimittu vain osa listaukseen 18.

```
1  const express = require('express')
2  const router = express.Router()
3  const Member = require('../../src/member')
4
5  const AWS = require('aws-sdk')
6  AWS.config.loadFromPath('./config.json')
7  AWS.config.apiVersions = {
8    dynamodb: 'latest'
9  }
10
11 const db = new AWS.DynamoDB.DocumentClient({region: 'eu-west-1'})
12 const member = new Member(db)
13 const bodyParser = require('body-parser').json()
14
15 router.post('/getMembers', bodyParser, (req, res, next) => {
16   member.getTeam(req.body.team, (error, success) => {
17     (error) ? res.send(error) : res.send(success)
18   })
19 })
20
21 router.post('/createMember', bodyParser, (req, res, next) => {
22   member.create(req.body, (error, success) => {
23     (error) ? res.send(error) : res.send(success)
24   })
25 })
26
27 router.post('/updateMember', bodyParser, (req, res, next) => {
```

Listaus 18: Osa aws.js-tiedostosta, joka hoitaa /aws-polkuun tulevan liikenteen reitityksen.

Listauksessa 19 otetaan tarkasteluun tiedoston määritykset. Rivillä 2 on esitetty reititin expressjs:n luokka, jonka avulla pystytään yksinkertaistamaan reitityksen toteuttamista koodissa. Rivillä 3 haettu Member-luokka toteuttaa sovelluslogiikan ja on täysin sama tiedosto kuin se, jota suorituspohjainen ratkaisu hyödynsi. Riveillä 5–9 alustetaan AWS-sdk-moduuli AWS DynamoDB -yhteyttä varten, joka luodaan rivillä 11.

```
1  const express = require('express')
2  const router = express.Router()
3  const Member = require('../../src/member')
4
5  const AWS = require('aws-sdk')
6  AWS.config.loadFromPath('./config.json')
7  AWS.config.apiVersions = {
8    dynamodb: 'latest'
9  }
10
11 const db = new AWS.DynamoDB.DocumentClient({reqion: 'eu-west-1'})
12 const member = new Member(db)
13 const bodyParser = require('body-parser').json()
```

Listaus 19: aws.js-tiedostossa tapahtuvat määritykset.

Listauksia 20 ja 21 vertaillessa voidaan todeta, että eri polkujen reitityksen eri member.js:n funktioille eivät juuri eroa toisistaan. Niissä otetaan vastaan polkuun tulevat viesti, parsitaan JSON-merkkijonoksi, JSON-objektiksi ja välitetään se funktiolle toteutettavaksi. Lopulta lopputuloksen perusteella palautetaan virhe- (error) tai onnistuminen-viesti (success). Kaikki toteutettavat funktiot ohjataan samantyylisillä reitityskomennoilla.

```
15 router.post('/getMembers', bodyParser, (req, res, next) => {
16   member.getTeam(req.body.team, (error, success) => {
17     (error) ? res.send(error) : res.send(success)
18   })
19 })
```

Listaus 20: /aws/getMembers-pyynnön reitit ys.

```
21 router.post('/createMember', bodyParser, (req, res, next) => {
22   member.create(req.body, (error, success) => {
23     (error) ? res.send(error) : res.send(success)
24   })
25 })
```

Listaus 21: /aws/createMember-pyynnön reitit ys.

5 Erot suorituspohjaiseen arkkitehtuuriin

Ratkaisuja on vertailtu kehitys- ja ylläpitotyön eroja vertailemalla projektien havaintojen perusteella. Taloudellisia eroja on pyritty arvioimaan Amazon Web Servicen hintojen perusteella tutkielman tekohetkellä, ja infrastruktuurin eroissa vertaillaan käytettyjä Amazon Web Service -resursseja. Toteutusta tehdessä huomattiin, että lähdekoodin tasolla eroja kahden eri ratkaisun välillä sovelluslogiikassa ei tarvinnut olla ollenkaan. Tämä tuli pienenä yllätyksenä aikaisempi kokemus huomioiden. Teknisten erojen vähäisyys ratkaisujen välillä oli hyvinkin yllättävää ja mielenkiintoista. Vertailussa keskitytään tarkemmin eroihin ratkaisujen välillä.

5.1 Kehitys- ja ylläpitotyön erot

Suorituspalveluarkkitehtuurin kehitysprosessi on hieman erilainen kuin perinteisellä arkkitehtuurilla. Esimerkiksi tutkielman tekohetkellä ei ollut mitään mahdollisuutta testata lambda-funktioita luotettavasti paikallisesti, joten käytännössä aina piti julkaista versio AWS-ympäristöön ilman, että toteutusta pääsi testaamaan. Perinteisemmän toteutuksen kanssa pystyy ajamaan täysin paikallisesti ratkaisua ja testamaan sitä, eikä sitä tarvitse viedä ennen valmista tuotosta oikeaan tuotantoympäristöön. Kehityksessä huomaa myös erot siinä, miten hallita tuotantoon vientiä, joka on todella tärkeä osa projektia. Serverless Framework on tarkoitettu myös tuotantoon vientiin, joten sen kanssa tilattomien funktioiden ja hajautetun sovelluslogiikan julkaiseminen on todella kivutonta. Perinteisemmässä mallissa ei ollut käytössä mitään julkaisutyökalua, koska sellaista ei tarvita kehitykseen. Tästä syystä julkaiseminen oli huomattavasti työläämpää. Kuitenkin myös perinteiselle mallille on tarjolla työkaluja, joiden avulla julkaiseminen on helpompaa, esimerkiksi Ansible ja Docker. Nämä julkaisutapojen erot on mahdollista siis vähentää yhteen komentoriviltä ajettavaan komentoon helposti, mutta perinteisen ratkaisun hyödyntäessä Ansiblea tai Dockeria, ylläpitotyö kasvaa verrattuna valmiiseen työkaluun, joka tekee julkaisun kehittäjän puolesta.

Ohjelmoinnin erot julkaisun lisäksi ovet oikeastaan vain pyyntöjä reitittävässä ja API-rajapinnan toteuttavassa osassa. Niissäkin se on vain teknologian määräämää. Listauksien 22 ja 23 välillä on vain vähän eroja. Listauksesssa 23 ei kerrota API-polkua, kun taas listauksessa 22 esitetään se. Lisäksi tilattoman toteutuksen listauksessa käytetään sisäänrakennettua callback-funktiota, joka pitää huolen vastauksen palauttamisesta kysyjälle. Perinteisen toteutuksen ratkaisussa se pitää

itse lähettää. Kun vieläpä molemmat ratkaisut hyödyntävät samaa sovelluslogiikkatiedostoa voidaan todeta, erot lähdekoodissa ovat hyvinkin vähäiset.

```
15 router.post('/getMembers', bodyParser, (req, res, next) => {
16   member.getTeam(req.body.team, (error, success) => {
17     (error) ? res.send(error) : res.send(success)
18   })
19 })
```

Listaus 22: Perinteisen ratkaisun getMember-reititys.

```
8  module.exports.get = (event, context, callback) => {
9    const t = (event.body === undefined) ? '*' :
       ↪  JSON.parse(event.body).team;
10   member.getTeam(t, callback);
11 };
```

Listaus 23: Suorituspohjaisen toteutuksen getMembers-reititys.

Ylläpidossa on myös merkittäviä eroja. Esimerkiksi sovelluksen toimivuuden ja saavutettavuuden valvonnnan suhteen. AWS tarjoaa valmiit työkalut, joilla seurata AWS Lambdojen ja AWS ApiGatewayn toimivuutta. Vain ylläpidon käyttöönotto on kehittäjän vastuulla. Työmäärä on vähäinen ja käyttöönotto on hyvin yksinkertaista. Perinteisessä ratkaisussa pitää toteuttaa valvonnat itse hyödyntäen saatavilla olevia työkaluja. Tämän jälkeen pitää toteuttaa näiden työkalujen valvonnat. Toisin sanoen ylläpito ja sovelluksen valvonta ovat jonkin verran työläämpiä perinteisessä mallissa kuin kilpailevassa ratkaisussa.

5.2 Testaamisen erot

Yksikkötestaamisen suhteen eroja ratkaisujen välillä ei ole, koska sovelluksen toiminnallisuus on täysin sama lähdekoodi ja sama sovelluslogiikka. Yksikkötestejä ei tarvinnut toteuttaa kuin apufunktioille Member.js-tiedostossa. Johtuen erojen

vähyydestä ja testien luonteesta ne esitellään yhdessä tässä vaiheessa. Apufunktiot ja niiden testit on esitelty listauksissa 24, 25, 26 ja 27.

```
224   filterExpired(teams) {
225     return teams.filter((member) => {
226       if (member.expires === 'always' || member.expires === '') {
227         return false;
228       } else {
229         return this.isExpired(member.expires.split('.'), new
          ↪  Date());
230       }
231     })
232   }
233
234   /*
235    * Check is date is expired
236    */
237   isExpired(memberDate, date) {
238     if (parseInt(memberDate[2]) < date.getUTCFullYear()) {
239       return true;
240     } else if (parseInt(memberDate[2]) === date.getUTCFullYear())
       ↪  {
241       if (parseInt(memberDate[1]) < date.getUTCMonth()) {
242         return true;
243       } else if (parseInt(memberDate[1]) === date.getUTCMonth()) {
244         if (parseInt(memberDate[0]) < date.getUTCDate()) {
245           return true;
246         }
247       }
248     }
249     return false;
250   }
```

Listaus 24: Member.js-tiedoston apufunktiot, osa 1.

```
270   createResponse(status, message, data) {
271     if (data === {}) {
272       return {
273         statusCode: status,
274         body: JSON.stringify({
275           "message": message
276         })
277       };
278     } else {
279       return {
280         statusCode: status,
281         body: JSON.stringify({
282           "message": message,
283           "data": data
284         })
285       };
286     }
287   }
```

Listaus 25: Member.js-tiedoston apufunktiot, osa 2.

```
292   filterTeams(data, team) {
293     let result = [];
294     data.forEach( (member) => {
295       if (member.memberships.length === 0 && team === '*') {
296         result.push(this.makeMemberRecord(member, {"team": "",
          ↪  "expires": ""}));
297       } else {
298         member.memberships.forEach( (membership) => {
299           if (membership.team === team || team === '*') {
300             result.push(this.makeMemberRecord(member,
              ↪  membership));
301           }
302         });
303       }
304     });
305     return result;
306   }
307
308   /*
309    * Will create member record for frontend.
310    */
311   makeMemberRecord(member, membership) {
312     return {
313       "email": member.email,
314       "name": member.name,
315       "phone": member.phone,
316       "address": member.address,
317       "team": membership.team,
318       "expires": membership.expires
319     }
320   }
```

Listaus 26: Member.js-tiedoston apufunktiot, osa 3.

```
10  describe('testing filter teams helper function', () => {
11    const teams = [{"email":"Mary@example.com", "name":"MaryDoe",
      ↪  "phone":"+358123456",
      ↪  "address":"Esimerkkistreet2,33100,TAMPERE", "memberships":
      ↪  [ {"team":"test","expires":"always"},
      ↪  {"team":"friba","expires":"30.03.2016"} ] },
12    { "email":"John@example.com", "name":"JohnDoe",
      ↪  "phone":"+358123456",
      ↪  "address":"Esimerkkistreet2,33100,TAMPERE", "memberships":
      ↪  [] }]
13
14    it('should list only team: test', () =>{
15      expect(member.filterTeams(teams,
        ↪  'test')).to.have.lengthOf(1)
16    });
17    it('should list all team', () => {
18      expect(member.filterTeams(teams, '*')).to.have.lengthOf(3)
19    });
20  });
21
22  describe('testing creating response', () => {
23    it('should return ok response', () => {
24      const tested = member.createResponse(200, "All okey!", {});
25      expect(tested).to.have.property('statusCode', 200)
26    });
27    it('should return ok response with data', () => {
28      const tested = member.createResponse(200, "All okey!",
        ↪  {"member":"isMember"});
29      expect(tested).to.have.property('statusCode', 200)
30      expect(tested).to.have.property('body',
        ↪  JSON.stringify({"message":"All okey!",
        ↪  "data":{"member":"isMember"}}))
31    });
32
33  });
```

Listaus 27: memberTests.js, osa 1 yksikkötestejä.

```
34
35    describe('testing create member record', () => {
36
37      it('should create member record', () => {
38          const m = { email: "testi@example.org", name: "test",
            ↪  phone: "0504003369", address: "Exmaple Streeet" }
39          const ms = { team: "APA", expires: "always" }
40          const r = { email: "testi@example.org", name: "test",
            ↪  phone: "0504003369", address: "Exmaple Streeet", team:
            ↪  "APA", expires: "always" }
41          expect(JSON.stringify(member.makeMemberRecord(m,
            ↪  ms))).to.equal(JSON.stringify(r));
42      });
43
44    });
45
46    describe('testing filter expired', () => {
47
48      it('should not contain JohnDoe', () => {
49        const teams = [{"email":"Mary@example.com",
          ↪  "name":"MaryDoe", "phone":"+358123456",
          ↪  "address":"Esimerkkistreet2,33100,TAMPERE",
          ↪  "team":"test","expires":"always" },
50        { "email":"John@example.com", "name":"JohnDoe",
          ↪  "phone":"+358123456",
          ↪  "address":"Esimerkkistreet2,33100,TAMPERE",
          ↪  "team":"test","expires":"01.01.1900" }]
51        expect(member.filterExpired(teams)).to.have.lengthOf(1)
52      });
53
54    });
55  });
```

Listaus 28: memberTests.js, osa 2 yksikkötestejä.

Palvelun toimivuuden testaaminen on hieman erilaista ratkaisujen välillä. Perinteisemmän sovellusarkkitehtuurin toteutus on helppo testata paikallisesti pyörittämällä sovellusta kehityskoneella, tai mahdollisesti esimerkiksi asentamalla se tuotantoympäristöä vastaavassa ympäristössä virtuaalikoneenassa paikallisesti. Vielä tutkielman toteutuksen aikana ei ollut tarjolla mahdollisuutta testata tilattoman ratkaisun koko toiminnallisuutta, mutta siihen on tulossa muutoksia muun muassa Amazon Web Servicen toimesta, AWS SAM Localin muodossa. AWS SAM Local vähentää tarvetta Serverless Frameworkiin, mutta vaikutuksia tilattoman ratkaisun kehitykseen tai testaamisen on vaikea arvioida ilman tutkimista [Hunt, 2017].

5.3 Taloudelliset erot

Tuloksien perusteella taloudellisten erojen arvioiminen suuressa mittakaavassa on hyvin vaikeaa. Ylläpitokustannuksia toteutetulle sovellukselle ei kummassakaan tapauksessa tule yhtään, minkä mahdollistaa AWS:n ilmainen suunnitelma [Services, 2017]. Ensimmäisen vuoden jälkeen kustannukset nousisivat todennäköisesti noin 4 dollariin kuukaudessa tilattomalle toteutukselle ja perinteisemmälle toteutukselle noin 5 dollariin kuukaudessa. Kustannukset tottakai nousisivat, mikäli palvelua käytettäisiin valtavia määriä. Käytön lisääntyessä tilattomille funktioille ei tarvitsisi periaatteessa tehdä juuri mitään, paitsi mahdollisesti lisätä hieman laskentatehoa, eli muistia, lambdoille tietokannan kasvaessa ja lisätä tietokantaan luku- ja kirjoitusoperaattorien määrää, jotta lambdat eivät estäisi toisiaan toimimasta. Perinteisemmässä toteutuksessa pitäisi kasvattaa AWS EC2 -instanssin kokoa ja harkita mahdollista rinnakkaisten instanssien lisäämistä. Kuitenkin tietokannan luku- ja kirjoitusoperaattoreita ei tarvitsisi lisätä kuin vasta rinnakkaisia ajoja lisätessä.
Taloudelliset erot kasvavat vielä entisestään, mikäli muutama API-rajapinnan kutsua käytetään todella paljon ja muita kutsuja selvästi vähemmän ja lähes marginaalisesti. Perinteisemmässä sovellusarkkitehtuuriratkaisussa palvelua voisi pilkkoa vielä pienempiin osiin eli muuttaa toteutusta hajautetuksi monoliitiksi. Tietenkään toteutuksessa oleva koodi ei vie resursseja, kun sitä ei ajeta. Monoliittisessä ratkaisussa on hyvin vaikea optimoida resursseja. Jos esimerkiksi olisi tunnistettavissa jokin käyttäjäryhmä, joka tekee valtavasti kyselyitä, ja suurin osa käyttäjistä ei tekisi sellaisia (ylläpitäjät versus peruskäyttäjät). Tällaisessa tilanteessa monoliitissa on hyvin vaikea lisätä vain ylläpitäjille resursseja, jolloin saattaa syntyä ylimääräistä kuluerää. Tämä johtaa siihen, että voidaan joutua

maksamaan resursseita, joita käytetään vähän ja harvoin, jotta peruskäyttäjillä on hyvä käyttäjäkokemus. Hajautettu monoliitti tarjoaisi tähän ratkaisun. Suorituspohjaisessa ratkaisussa resursseja ei tarvitse varata vähäisellä käytöllä olevilla osille, koska jokainen tilaton funktio käynnistetään vain tarvittaessa. Sillon kuin kukaan ei käytä palvelua niin resurssien rasitus on automaattisesti nolla.

5.4 Ratkaisujen infrastruktuuriset erot

Merkittävimmät erot toteutusten välillä ilmenivät teknologioissa, joita taustalla käytetään. Ratkaisujen käyttämien palveluiden välillä on merkittäviä eroja, vaikka molemmat ratkaisut tukeutuvat AWS DynamoDB:hen tietovarastona ja sovelluslogiikan toteuttaa sama lähdetiedosto. Esimerkiksi tilattoman ratkaisun jokainen API-rajapinnan kutsu käynnistää oman muista kutsuista irrallisen toiminnon, joka suorittaa halutun asian. Näitä toimintoja voi olla periaatteessa rajaton määrä käynnissä samaan aikaan, tietenkin tietokannan kirjoittajat ja lukijat rajoittavat sitä, kuinka moni voi olla yhteydessä tietokantaan samaan aikaan. Vastaavanlaisesti perinteisessä mallissa virtuaalikoneen resurssit rajoittavat suorituskykyä jo ennen kuin päästään tietokannan rajoituksiin, tosin myös tässä mallissa tietokannan rajoitteet ovat mahdollisena ongelmana.
Yksi huomattava ero on myös se, että AWS Lambda -toteutuksessa API-rajapinta on käytännössä eriytetty sovelluslogiikasta täysin, koska sen toteuttaa AWS ApiGateway. Serverless Framework piilottaa toteutuksen eriytyksen, koska AWS Lambdojen ja ApiGatewayn resurssien luontia hallitaan samasta kohtaa tiedostoa. Eriytys mahdollistaa esimerkiksi sen, että osan API-rajapinnasta pystyy toteuttamaan helposti muilla keinoin, tai ohjata suoraan toiseen palveluun eriyttämisen kautta. Esimerkiksi mikäli haluttaisiin osa vanhasta perinteisellä mallilla toteutetusta taustapalvelusta korvata Lambdoilla, voidaan reititys ei vielä korvattuihin polkuihin tehdä AWS ApiGatewayn kautta ilman lambda-funktioitakin. Näin voidaan hyvin yksinkertaisesti piilottaa loppukäyttäjiltä API-rajapinnassa tapahtuvat muutokset.

6 Yhteenveto

Yksi mielenkiintoisimpia tuloksia tutkielmassa on perinteisen monoliittisen taustasovelluksen ja hajautettuun staattisiiin funktioihin erotetun ratkaisun lähdekoodien vähäinen eroavaisuus. Työmäärä perinteisen NodeJS-sovelluksen muuttamisen AWS Lamdoilla toteutetuksi on hyvinkin pieni, mikäli sovellus on alunperin toteutettu tilattomaksi siten, että ainoa tilanhallinta tapahtuu tietokannassa. Nykypäivänä kaikki pilvipalvelut tuleekin toteuttaa tilattomina, jotta automaattiskaalautuminen voidaan toteuttaa perinteisen sovellusarkkitehtuurin kanssa. Sovellusten kehityksessä merkittävä osa ajasta eli sovelluslogiikan toteuttaminen on siis käytännössä saman mittainen. Esimerkiksi tutkielmassa toteutetut versiot tehtiin ensin Lambda-toteutuksena ja tämän jälkeen muutamassa tunnissa toteutettiin Expressjs-reititys ja perinteisemmän arkkitehtuurin mukainen taustapalvelusovellus oli toteutettu. Toteutusten haastavuudessa teknisellä tasolle ei myöskään ole suuria eroja. Tietenkin ongelmaksi voi muodostua AWS Lambdojen tilattomuus, jolloin joitakin perinteisiä ratkaisuja ei voi hyödyntää tai jakaa resursseja eri Lambda-kutsujen välillä.

Valvonnassa ja ylläpidossa havaittiin merkittävimmät erot ja näiden kohdalla Lambda-toteutus tuntuu olevan selkeä voittaja. Työmäärä valvontojen toteuttamiseksi on hyvinkin pieni verrattuna perinteisempään virtuaalikoneratkaisuun. Tosin samalla ylläpitäjä luottaa merkittävän osan valvonnasta ja ylläpidosta Amazon Web Servicen hartioille. Varsinkin pienemmille toimijoille vastaava malli on hyvinkin houkutteleva johtuen ylläpidettävien järjestelmien vähäisyydestä. Vaikka itse julkaisun saa toteutettua molemille ratkaisulle yhtä kevyesti hyödyntämällä jatkuvan kehityksen työkaluja, kuten Ansiblea perinteisen mallin kanssa, niin Lamdojen yhteydessä käytetty Serverless Framework tekee kehittäjän puolesta hieman enemmän asioita ja tarjoaa hieman turvallisemman ympäristön kehittäjän omien virheiden välttämiseen täysin valmiilla työkalulla.

Taloudellisia eroja varsinkaan pienemmissä toteutuksissa ei juurikaan pääse ilmenemään, mutta AWS Lambda -toteutuksen etuna voidaan pitää täydellistä skaalautumista suuremmalle kuormalle ilman mitään työvaiheita. Tämän ominaisuuden lisäksi myös kuormattomina aikoina kulujen väheneminen on taloudellisesti merkittävä etu. Monesti suuren kuorman palveluiden kohdalla perinteisemmän mallin toteutuksiin pitää konfiguroida jonkinlainen automaattinen skaalautuminen, jotta suuren kuorman tilanteista selvitään ilman ongelmia ja vähäisellä kuormalla ei käytetä turhaan maksullisia resursseja.

Tutkielmassa tarkasteltiin hyvin pitkälti JavaScriptin näkökulmasta toteutusten

teknisiä eroja. AWS:n Lambdat tukevat myös Java (8)-, Python- ja C# (.Net core) -ohjelmointikieliä, joita ei tässä tutkielmassa tarkasteltu. Erot perinteisemmän taustapalvelusovelluksen ja lambda-toteutuksen välillä voivat olla merkittävämmät muilla kielillä. Kuitenkin esimerkiksi Pythonin kohdalla perusperiaate pysyy samanlaisena ja mikäli sovelluslogiikka on toteutettu erillisena kutsujen reitityksestä, niin esimerkiksi tiedon lisääminen tietokantaan ja sen käsittely on samanlaista ja oikein toteutettuna samaa koodia.

Viiteluettelo

[Amazon, 2015] Amazon. https://d0.awsstatic.com/whitepapers/AWS_Serverless_Multi-Tier_Architectures.pdf, 2015. [Checked 02-06-2017].

[Amazon, 2016] Amazon. https://d0.awsstatic.com/whitepapers/AWS_Cloud_Best_Practices.pdf, 2016. [Checked 02-06-2017].

[Amazon, 2017a] Amazon. Amazon Web Services. https://aws.amazon.com/, 2017. [Checked 02-06-2017].

[Amazon, 2017b] Amazon. AWS API Gateway. https://aws.amazon.com/api-gateway, 2017. [Checked 01-05-2017].

[Amazon, 2017c] Amazon. AWS CloudWatch. https://aws.amazon.com/cloudwatch, 2017. [Checked 01-05-2017].

[Amazon, 2017d] Amazon. AWS DynamoDB. https://aws.amazon.com/dynamodb, 2017. [Checked 01-06-2017].

[Amazon, 2017e] Amazon. AWS IAM. https://aws.amazon.com/iam/, 2017. [Checked 02-06-2017].

[Amazon, 2017f] Amazon. AWS Lambda. https://aws.amazon.com/lambda, 2017. [Checked 01-05-2017].

[Amazon, 2017g] Amazon. AWS Lambda — Pricing. https://aws.amazon.com/lambda/pricing/, 2017. [Checked 01-05-2017].

[Amazon, 2017h] Amazon. AWS S3. https://aws.amazon.com/S3, 2017. [Checked 02-06-2017].

[Google, 2017] Google. Google Cloud Platform. https://cloud.google.com/, 2017. [Checked 02-06-2017].

[Hastings, 2013] R. Hastings. *Making the Most of the Cloud: How to Choose and Implement the Best Services for Your Library*. Scarecrow Press, 2013. [Online] Checked: http://dx.doi.org/10.1007/s00607-013-0346-9.

[Hendrickson *et al.*, 2016] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, & Remzi H Arpaci-Dusseau. Serverless computation with openlambda. *Elastic*, 60:80, 2016.

[Hunt, 2017] Randall Hunt. AWS Sam Local. `https://aws.amazon.com/blogs/aws/new-aws-sam-local-beta-build-and-test-serverless-applications-locally/`, 2017. [Checked 11-08-2017].

[IBM, 2017] IBM. IBM Bluemix. `https://www.ibm.com/cloud-computing/bluemix/`, 2017. [Checked 02-06-2017].

[Microsoft, 2017] Microsoft. Microsoft Azure. `https://azure.microsoft.com/`, 2017. [Checked 02-06-2017].

[Mochajs, 2017] Mochajs. Mochajs. `https://mochajl.readthedocs.io/en/latest/`, 2017. [Checked 15-06-2017].

[Roberts, 2016] Mike Roberts. Serverless Architectures. `https://martinfowler.com/articles/serverless.html`, 2016. [Checked 06-04-2017].

[serverless.com, 2017] serverless.com. serverless.com. `https://serverless.com/`, 2017. [Checked 28-04-2017].

[Services, 2017] Amazon Web Services. AWS Free Tier. `https://aws.amazon.com/free/`, 2017. [Checked 14-08-2017].

[Spillner, 2017a] J. Spillner. Exploiting the Cloud Control Plane for Fun and Profit. *ArXiv e-prints*, January 2017.

[Spillner, 2017b] J. Spillner. Snafu: Function-as-a-Service (FaaS) Runtime Design and Implementation. *ArXiv e-prints*, March 2017.

[StrongLoop/IBM, 2017] StrongLoop/IBM. Expressjs. `https://expressjs.com/`, 2017. [Checked 28-04-2017].

[Technologies, 2017] Postdot Technologies. Postman. `https://www.getpostman.com/docs/`, 2017. [Checked 14-07-2017].

[Walraven *et al.*, 2013] Stefan Walraven, Eddy Truyen, & Wouter Joosen. Comparing paas offerings in light of saas development. *Computing*, 96(8):669–724, 2013. [Online] Checked: http://dx.doi.org/10.1007/s00607-013-0346-9.

[Yan *et al.*, 2016] Mengting Yan, Paul Castro, Perry Cheng, & Vatche Ishakian. Building a chatbot with serverless computing. In *Proceedings of the 1st International Workshop on Mashups of Things and APIs*, MOTA '16, pages 5:1–5:4, New York, NY, USA, 2016. ACM.